ESD-TR-66-113

MTR - 197

A METHOD FOR THE EVALUATION OF SOFTWARE

VOLUME II

Procedural Language Compilers -- Particularly COBOL and FORTRAN

APRIL 1967

A. E. Budd

Prepared for

EDP EQUIPMENT OFFICE
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 8510

Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract AF19(628)-5165

ABSTRACT

This report contains procedural language compiler features considered important for comparative analysis. These features are identified in a form expressly for inclusion in the Three Step Method for Software Evaluation (Volume 1 of this series) under Category ONE: Procedural Language Compilers -- Particularly COBOL and FORTRAN.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

S. P. STEFFES
Colonel, USAF
Chief, EDP Equipment Office

TABLE OF CONTENTS

TABLE OF CONTENTS (CONCLUDED)

SECTION ONE

INTRODUCTION

This report contains the important features of procedural language compilers, particularly COBOL and FORTRAN. The compiler features identified are those which have been known to vary significantly from vendor to vendor, and are easily determined by appropriate questioning or actual demonstration. These features are a composite from which the evaluator, in conjunction with the user, may select providing they are consistent with the system requirements. The evaluator would first select those features which are of significant importance to the user. This list would be part of the basic system requirements of the evaluation team in which one vendor's answer or result for one feature would form one entry. This would provide a convenient method of comparing the features of one vendor as well as comparing the vendor's response to any one feature.

The evaluator must work with the user to determine his specific needs. For instance, the user may require emphasis on compilation since the major activity is to develop experimental programs which will be executed only a few times and discarded. On the other hand, his needs may require a very efficient object code from the compiler because once the program has been checked out it will be executed many times per day for several years with only slight modifications. Whichever the case may be, these needs must be determined through the user so that the resulting system selection will meet his specific requirements.

Any evaluation must begin with a precise identification of the user requirements. Failure to determine user requirements with sufficient precision can lead to identification of a grossly unsuitable system as the optimal one. Two types of user requirements must be identified. First, features of possible systems must be examined to determine their utility to the user; in the initial stages of the evaluation process this utility need not take the form of a precise weight but need only be determined sufficiently to select the subset of possible features to be analyzed during the evaluation process. Once the proper group of features has been selected, a standard of quality must be set for each one. This requires that a minimum level of performance regarding each feature be established. Obviously, instances will occur in which the presence or absence of the feature

is the only matter of concern, but in the case of certain features particularly amenable to quantitative measurement a minimum performance level together with some valuation on above minimum performance must be established.

The features are presented in three sections. The first of these, SECTION TWO, contains summary titles of each feature in the matrix form required for the three step procedure described in Volume I of this series. The next section, SECTION THREE, contains a description of what is meant by each summary titled feature, which measures of software capability are affected by this feature and, where necessary, further explanation indicating what aspects are especially useful to an EDP installation or specific application. The last section, SECTION FOUR, contains a questionnaire which may be used by evaluator personnel to determine what important features are prevalent relative to vendor's COBOL or FORTRAN compilers.

It should be remembered that these features and questions are a composite and therefore, only those features or questions deemed important relative to the specific application or installation under consideration should be selected.

Each section is presented in the following seven groups:

I. Source Language Oriented Features - those relating primarily to characteristics.

II. Logic and Control of Input/Output Devices - features referring specifically to the control of input/output devices and associated program logic (buffer control, device assignment, etc.).

III. Compiler Features - features of the processor (i.e. the implementation of the language) rather than the language itself.

IV. Programmer Error Diagnostics - features relating to the facilities for error detection.

V. Hard-Copy Output and Facility Documentation - characteristics relating to the nature and frequency of documents produced by the compiler system (listings, error reports, etc.) as well as to available documentation about the processor and the language.

VI. Manual and Automatic Operations - compiler and object system characteristics pertaining to running the compiler, updating to produce new compiler versions, etc.

VII. Facility Administration - matters pertaining to the administration of a facility using the software package in question as well as those not conveniently included under any other category.

SECTION TWO

MATRIX OF FEATURES

The COBOL language is oriented toward business data-processing problems which involve manipulation of files of data. Business problems are represented by a relatively small amount of algebraic and logical processing. Instead, considerable emphasis is placed on manipulation of large files of basically similar records. The algebraic compilers used in scientific calculation (FORTRAN, ALGOL, etc.) require emphasis on the procedures involved in a complex mathematical problem using numerical approximation, complicated logical structures, etc. In fact, in FORTRAN, all calculations at the highest level are performed on floating point numbers or integers each requiring a full word in word organized machines. Any data packing must be performed at the machine language level. On the other hand, in COBOL the source language level provides the means to structure data items in records and working storage in a very flexible way. Furthermore, there are rewards in program running time for the programmer who carefully considers different methods of structuring the items in the DATA DIVISION and WORKING STORAGE. To do this he should know the word lengths and other unique features of this particular computer hardware, but he implements his knowledge at the source language level rather than at the machine language level. These differences require that features of COBOL will exhibit emphasis mainly in the description and manipulations of (1) data items, and (2) input-output record handling.

Each feature designated as important for comparative compiler analysis has been identified and placed in this section with the format appropriate for inclusion in the three step procedure described in ESD-TR-66-113, Volume I. Features which reference COBOL are ones which are applicable only to COBOL; similarly, FORTRAN referenced features apply only to FORTRAN. Those having no language reference apply to COBOL, FORTRAN and other procedural language compilers.

The character "X" has been used to show which measures of software capability are affected by the features. It is apparent that any one feature may, in some esoteric way, affect each and every measure of software capability; however, an "X" has been placed under those measures which significantly affect the feature in question. The user-evaluator team must determine what measures are of particular importance to them, in addition to deciding whether each measure will be affected favorably or unfavorably (and to what degree) according to the system requirements. This will establish ground rules such that proposals may be evaluated relative to preestablished specifications appropriately adjusted to reflect characteristics of the application.

REQUIREMENT	GENERAL MEASURES OF SOFTWARE CAPABILITY / PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN I. SOURCE LANGUAGE ORIENTED FEATURES	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Source Language Dump & Trace Requests		X	X											
	B. Adequate I/O Source Language Facilities	X	X			X									
	C. Available Languages Other Than Required								X						
	D. COBOL - Edition 1965	X											X		
	E. COBOL - Additional Features	X											X		
	F. COBOL - I/O Record Size					X	X								
	G. COBOL - Repeated Data	X													
	H. Effects of Non-Standard Implementation			X	X										
	I. FORTRAN - ASA FORTRAN Standard Features	X											X		
	J. FORTRAN - Features Not Equivalent to ASA	X											X		
	K. FORTRAN - DO Statement Indexing Differences						X								

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN I. SOURCE LANGUAGE ORIENTED FEATURES (Continued) GENERAL MEASURES OF SOFTWARE CAPABILITY	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	L. FORTRAN - DO Subscripts and Limit Differences	X											X		
	M. FORTRAN - Frequency Statement Provision						X								

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN / GENERAL MEASURES OF SOFTWARE CAPABILITY II. LOGIC AND CONTROL OF INPUT/OUTPUT DEVICES	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Simultaneous Reading/Writing of I/O Files				X	X				X					
	B. Simultaneous Buffering of I/O Files				X	X				X					
	C. Double Buffering of I/O Records				X	X									
	D. Utilization of Priority Interrupt					X			X						
	E. Options for I/O Device Servicing	X					X								
	F. Advantages of I/O Device Servicing Options	X					X								
	G. Restrictions on Computation and I/O Overlapping				X	X				X					
	H. I/O Units Activated Concurrently				X	X				X					
	I. Repeated Tries on I/O Device Errors				X	X									
	J. Method of Supplying I/O Routines to Object Program					X	X								
	K. I/O Subroutines Always Loaded						X								

CATEGORY ONE: PROCEDURAL LANGUAGE COMPILERS

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN / GENERAL MEASURES OF SOFTWARE CAPABILITY II. LOGIC AND CONTROL OF INPUT/OUTPUT DEVICES (CONTINUED)	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	L. COBOL - Number of I/O Areas Supplied for RESERVE					X										
	M. COBOL - Action Taken When Buffer Holds Several Records				X	X										
	N. Utility of Random Access I/O Devices				X											
	O. COBOL - Record Selection on Random Access				X											
	P. COBOL - Random Access Storage for Test Data		X													

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN III. COMPILER FEATURES / GENERAL MEASURES OF SOFTWARE CAPABILITY	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Special Patching Language Provided		X	X										X	
	B. Source Language Level Patching		X												
	C. Parameters in Design Point						X		X						
	D. Disabling of Time-Efficiency Trade-Offs						X								
	E. Efficiency of Object Code				X		X								
	F. Separation of Common Subfunctions				X		X								
	G. Procedure for Running Large Programs (Segmentation)	X													
	H. Common Data References for All Programs	X													
	I. Modification of Program Control Sequence Dynamically	X													
	J. Automatic Library Search	X													
	K. Standard Record and File Description												X		

REQUIREMENT	GENERAL MEASURES OF SOFTWARE CAPABILITY / PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN III. COMPILER FEATURES (CONTINUED)	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	L. Compiler Table Sharing for Overflow						X	X							
	M. Dynamic Allocation of Storage			X				X							
	N. Implementation of Storage Allocation Changes										X				
	O. Hardware Configuration Changes at Program Running Time		X		X										
	P. Combining Program Segments and Routines from Other Software Packages	X	X										X		
	Q. Operation on Similar Machine Types												X		
	R. Utilization of Bit or Byte Accessing Facilities and Other Sophistications				X		X								
	S. Packing Techniques for Conserving Storage						X	X							
	T. Truncation or Rounding of Conversions	X													
	U. Subroutine Names Considered as Global Symbol	X			X										
	V. Common Subexpressions Compiled Only Once						X								

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN III. COMPILER FEATURES (CONTINUED) / GENERAL MEASURES OF SOFTWARE CAPABILITY	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	W. Mathematically Equivalent Subexpressions						X								
	X. Modification of Constants		X	X											
	Y. Intermediate Level Languages Used	X	X										X		
	Z. Programmer Selection of Outputs	X													
	AA. Target Language Optimization Performed			X			X								
	AB. Ease of Excluding Optimization			X											

REQUIREMENT	GENERAL MEASURES OF SOFTWARE CAPABILITY / PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN IV. PROGRAMMER ERROR DIAGNOSTICS	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Errors Causing Compilation Termination			X	X							X			
	B. Undetected Source Language Errors		X												
	C. Errors Causing Unexpected Stops				X							X			
	D. Other Source Program Errors		X	X											
	E. Number of Errors Found Before Termination		X	X											
	F. Number of Passes Required for Errors	X		X											
	G. Compiler Production After Error Detection			X											
	H. Statement Identification for Errors		X												
	I. Detection of Excessive Source Language Symbols		X	X											
	J. Errors Detected During Execution		X		X										
	K. Modifications for Trace Requests		X		X										

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN IV. PROGRAMMER ERROR DIAGNOSTICS (CONTINUED) GENERAL MEASURES OF SOFTWARE CAPABILITY	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	L. Prescan for Frequent Error Detection		X	X											

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN V. HARD-COPY OUTPUT AND FACILITY DOCUMENTATION / GENERAL MEASURES OF SOFTWARE CAPABILITY	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Diagnostics Provided Off-Line		X											X	
	B. Automatic Update of Recompilations			X										X	
	C. Trace and Dump Outputs Printed		X												
	D. Trace Symbols and Statement Referencing		X												
	E. Accounting for Compiler Modifications								X					X	
	F. Documentation Intended for External Maintenance										X			X	
	G. Availability of Compiler Manuals and Listings														
	1. Listings of the Compiler										X				
	2. Manual of Compiler Operation		X								X			X	X
	3. Manual of Compiler Internal Structure										X				
	4. Programmer's Reference Manual	X												X	X

- THREE STEP METHOD FOR SOFTWARE EVALUATION -

CATEGORY ONE: PROCEDURAL LANGUAGE COMPILERS

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN / GENERAL MEASURES OF SOFTWARE CAPABILITY V. HARD-COPY OUTPUT AND FACILITY DOCUMENTATION (CONTINUED)	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	5. Programmer Training Manual	X													X
	H. Outputs of Compilation and Loading														
	1. A Listing of the Source Program		X											X	
	2. A Listing of the Object Program		X												
	3. Cross-references to Symbols and Statements		X												
	4. A Listing of Data Areas and Their Locations		X											X	
	5. A Listing of Buffer Areas and Their Locations		X												
	6. A Storage Map Including Routines Loaded		X											X	
	7. Lists of All Inter-program References		X												
	8. A List of Detected Errors		X												

REQUIREMENT	GENERAL MEASURES OF SOFTWARE CAPABILITY / PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN VI. MANUAL AND AUTOMATIC OPERATIONS	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Operator Message for Unavailable I/O Files											X			
	B. Operator Messages for Object Program Errors											X			
	C. Machine Error System Termination											X			
	D. Operator Options for Termination											X			
	E. Information for I/O Device Malfunction					X						X			
	F. Failure Reports for Maintenance Personnel										X				
	G. Operator's Console Control of Trace Routines		X									X			
	H. Programs for Updating Software System										X				
	I. Minimum Configuration for Updating										X				
	J. Operator Communication Via Console Keys											X			
	K. Time Required for Operator Actions			X	X							X			

- THREE STEP METHOD FOR SOFTWARE EVALUATION -

CATEGORY ONE: PROCEDURAL LANGUAGE COMPILERS

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN / GENERAL MEASURES OF SOFTWARE CAPABILITY VI. MANUAL AND AUTOMATIC OPERATIONS (CONTINUED)	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	L. Executive or Monitor Control Provision			X	X							X			
	M. Availability of Compile, Load and Execute Options			X	X							X			
	N. Flexibility of Compile, Load and Execute Options	X													

REQUIREMENT	PROCEDURAL LANGUAGE COMPILERS--PARTICULARLY COBOL AND/OR FORTRAN / GENERAL MEASURES OF SOFTWARE CAPABILITY VII. FACILITY ADMINISTRATION	PROGRAMMING	CHECKOUT	COMPILATION	EXECUTION	I/O UTILIZATION	PRODUCT ECONOMY	SECONDARY STORAGE	GROWTH FLEXIBILITY	SIMULTANEITY	MAINTENANCE	INTERVENTION	INDEPENDENCE	DOCUMENTATION	TRAINING
	A. Same Compiler Version Used for Both Timing and Documentation		X								X				
	B. Maintenance Performed by Vendor or User										X				
	C. Stability of Compiler Design										X				
	D. Unusual Techniques of Implementation			X	X						X				
	E. Size of Machine Language Version			X				X							
	F. Use of Standard Modules Within Compiler								X		X				
	G. Minimum Configuration Required for Compiler	X		X											
	H. Performance Improvements from Larger Configurations								X						
	I. Average Speed of Compiler (Statements/Minute)			X											
	J. Object Program Storage Space Used for Overhead, Buffers, Monitor Code, etc.						X	X							
	K. Degree of Utilization of Command Structure in Object Code						X								

SECTION THREE

DESCRIPTIONS OF PROCEDURAL LANGUAGE COMPILER FEATURES

I. SOURCE LANGUAGE ORIENTED FEATURES

A. Source Language Dump and Trace Requests

This refers to the capability of inserting dump and control flow trace requests at the source language level. Can the programmer insert these types of requests without modifying his program, and can they be specified at the source language level? Can this routine be turned on and off externally from the console or by a single operating system command? Are outputs printed in source language? If not, how are the source statements and symbols referenced to the output? Appropriate use of this feature decreases program checkout time since the makeup of checkout runs and multiple branch test runs is significantly simplified. Some compilers may become more complex with this capability which may affect compilation time.

B. Adequate I/O Source Language Facilities

The capability of the source language statements or facilities to provide access to all types of input/output equipment available for the machine is a most desirable feature and in some cases an absolute necessity. Can the programmer use all types of input/output equipment available for the proposed machine? Can the hardware configuration be changed easily for the program at object running time? Is there a source language statement or suitable subroutine access through the source language for each different type of I/O device? Is there sufficient flexibility to access each available device within a single type? Is the structure of the compiler sufficiently flexible to provide for quick addition of future new I/O devices or types of devices? This would decrease programming and checkout time since the programmer would not be required to produce the machine language instructions and associated checkout each time the individual I/O units were first used in a program. It would also enhance I/O utilization.

C. Available Languages Other Than Required

Other language systems which are available at essentially no increased cost may prove useful for some future requirement. Past experience has shown that installations very often find their requirements change including a need for new areas of application. What

languages are available on the proposed configuration? Is each operative now or at installation delivery? Is each well documented? Additional languages would contribute to growth flexibility.

D. COBOL - Edition 1965

What is the degree of compatibility to which this language meets the requirements and recommendations laid down by the CODASYL Executive Committee in the report titled "COBOL, Edition 1965, Department of Defense"? What features of this possible standard have not been included in the proposed compiler? Compliance in this area will provide for decreased programming time as well as contribute measurably to the degree of machine independence.

E. COBOL - Additional Features

Describe any features this compiler has which provide increased capability or flexibility which are not contained in COBOL, Edition 1965. Evaluation is performed on how well each feature contributes to decreasing programming time and increasing machine independence.

F. COBOL - I/O Record Size

Particular software or hardware features of some computing systems may limit the variability in data record sizes. The flexibility available for varying the size of input/output records is a primary concern. Describe the options available in the SIZE clause relative to Format 1 and Format 2 as stated in the COBOL, Edition 1965, document. This feature will tend to increase the efficiency of I/O utilization as well as to produce a more efficient object program.

G. COBOL - Repeated Data

Some computer applications utilize the ability to generate tables of repeated data more than others. The OCCURS clause provides a method of generating repeated data and supplies information for implementing of subscripts and indices. Describe the options available in the OCCURS clause relative to Format 1 and Format 2 as stated in the COBOL, Edition 1965, document. This feature will tend to reduce programming time.

H. Effects of Non-Standard Implementation

It may occur that certain compilers will have features which are described as standard but whose implementation is measurably different than intended by that report. Any significant differences

relative to this implementation must be assessed and compared relative to this procedural language. Describe the effects of each non-standard implementation scheme used. How does each scheme affect compilation time and execution time? Techniques could be conceived which tend to decrease these times and others which tend to increase them. Care must be taken to assess which situation prevails and to what degree.

I. FORTRAN - ASA FORTRAN Standard Features

List and describe all features of the vendor's proposed language which are equivalent to some feature of the ASA FORTRAN (proposed) standard. List and describe all features included in ASA FORTRAN (proposed) but not included in the vendor's proposed language. Appropriate use of each feature will tend to decrease programming time as well as contribute to machine independence.

J. FORTRAN - Features Not Equivalent to ASA

List and describe all features of your language which have no equivalent in the proposed ASA FORTRAN standard. Each should be described relative to any increased capability or flexibility offered. Evaluation is performed on how well each feature contributes to decreasing programming time and increasing machine independence.

K. FORTRAN - DO Statement Indexing Differences

How do the restrictions on statement sequencing differ from those described in the proposed ASA FORTRAN standard? Some FORTRAN systems rearrange statements appearing in the range of a DO so that those statements not dependent on the DO index are removed from the range of the DO and placed before it. This is a highly desirable feature since the statements removed will be executed only once instead of once for every iteration. However, the algorithms used to perform this rewriting must be sufficiently sophisticated to allow for the fact that dependence on the index value may occur at several levels. Appropriate inclusion of this feature tends to produce an economical object program.

L. FORTRAN - DO Subscripts and Limit Differences

How do the restrictions on subscript expressions and on expressions used as the limits of a DO differ from the proposed ASA FORTRAN standard? Flexibility with respect to this feature will decrease programming time and increase machine independence.

M. FORTRAN - FREQUENCY Statement Provision

Is the FREQUENCY statement accepted by the proposed compiler? If so, describe the effects of the FREQUENCY statement. Its purpose is to enable the compiler to produce more efficient coding of conditional statements by specifying the expected relative frequencies of proceeding along each branch. Programs could be compiled with and again without FREQUENCY statements. Comparing the resulting object programs would provide an appropriate test of this facility.

II. LOGIC AND CONTROL OF INPUT/OUTPUT DEVICES

A. Simultaneous Reading/Writing of I/O Files

If the proposed hardware permits, can several input/output files be read or written simultaneously using the proposed software? Identify the numbers of files which may be read simultaneously using the limits of the software system. Identify the number of files which may be written simultaneously with the proposed software. List and describe those input, compute and output combinations of simultaneity available with this software package.

B. Simultaneous Buffering of I/O Files

This feature refers to the capability for simultaneous buffering of files. Identify to what degree input files and output files may be buffered simultaneously. Are different buffer areas used for each simultaneous data transfer?

C. Double Buffering of I/O Records

Is double buffering of input/output records provided? That is, are there two or more distinct areas in storage for input or output transfer? Can the system be reading from one and computing with a second? Can it be writing from one and computing with the second. Determination should be made as to the flexibility of this scheme. This affects the I/O utilization software measure of capability.

D. Utilization of Priority Interrupt

If the proposed configuration includes a priority interrupt system, does the proposed software system utilize this priority interrupt for input/output processing? In other words, does the software system take advantage of the hardware in the area of priority interrupts? If not, what must the user do in order to implement this in his own system? This feature will increase the degree of I/O utilization and may also contribute to growth flexibility.

E. Options for I/O Device Servicing

Describe options available in the area of input/output device servicing. Does the proposed compiler allow the user to directly effect tape rewinding, end of file placement, setting up or clearing of sections of drums or discs? Identify each option. Are these functions performed automatically by the software system. If so, is there sufficient flexibility left to perform the user requirements?

F. Advantages of I/O Device Servicing Options

What are the advantages and flexibilities (in terms of ease of programming, economy in the object program, etc.) of the above options? Identify the reasons for making the options available and relate each to the advantages and flexibilities.

G. Restrictions on Computations and I/O Overlapping

What are the restrictions on overlapping computation with input/output operations? Which of these restrictions are due to software rather than hardware? Can computation be overlapped with reading? Can computation be overlapped with writing? Can computation be overlapped with reading and writing simultaneously? This feature will affect I/O utilization and simultaneity.

H. I/O Units Activated Concurrently

What combinations of input/output units may be concurrently activated? What input/output units may not be activated concurrently? Identify each input/output unit and its capability of being activated concurrently with the proposed hardware configuration.

I. Repeated Tries on I/O Device Errors

On which input/output devices will attempts to read or write be repeated if errors are detected? For example, redundancy calculations on certain magnetic tape units should be tried several times before giving up. What is the basis for choosing the given number of repeated tries on each device? Has it been shown that each device has sufficient past history to warrant these repeated tries? Does the compiler provide suitable messages to both the computer operator and the programmer in each case? Is the maintenance function aware of each repeated try? Identify each detected error relative to the proposed configuration in addition to describing the repeated procedure. This feature will tend to optimize I/O utilization and to a lesser degree will afford some savings in execution time.

J. Method of Supplying I/O Routines to Object Programs

How are the input/output subroutines supplied to the object program? Are they organized by device, by the function they perform? Identify the cases where each subroutine is provided to the object program. This feature will tend to increase product economy and affect the flexibility of I/O utilization.

K. I/O Subroutines Always Loaded

Which input/output subroutines are always loaded with the object program? In some compiler systems input, or output (or both) are loaded whether they are needed or not. If they are not needed they will take up space and thereby reduce the product economy. If a particular device is needed in the program under consideration, only for reading input to this program, is the output routine also loaded? On the other hand if the output routine is needed is the input routine loaded automatically? This feature is useful for comparative analysis since product economy is affected. Input and output routines are characteristically large and in many cases are not needed in all their flexibility. It would be desirable to have a minimum, in some cases, input or output routine to provide transfer of a small number of parameters in an unsophisticated format in addition to the normal large complex input/output routines.

L. COBOL - Number of I/O Areas Supplied for Reserve

How many input/output areas have been implemented under the RESERVE option of the FILE-CONTROL statement, INPUT-OUTPUT Section of the ENVIRONMENT DIVISION?

M. COBOL - Action Taken When Buffer Holds Several Records

When available buffer areas can hold more than one record, what special action is taken by the system? Does the program make suitable adjustments to take advantage of the increased storage space? Program execution time will be saved and I/O utilization increased when larger quantities of data are read from and/or written to files.

N. Utility of Random Access I/O Devices

What provisions have been implemented in this compiler to provide the programmer with facilities to take advantage of these random access devices? This feature will tend to decrease execution time of file manipulating programs which previously were tape oriented.

O. COBOL - Record Selection on Random Accesses

Does the compiler allow the programmer to use the random access device for file storage? If so, can any single record be selected by a read or write verb? Is there sufficient flexibility to allow the programmer to get any random record in the response time afforded by the device?

P. COBOL - Random Access Storage for Test Data

If the proposed configuration includes a separate random access device is this random access device used for test data storage during debugging runs? To what extent?

III. COMPILER FEATURES

A. Special Patching Language Provided

Is there a special patching language provided? In some compilers a facility has been made available which will allow the patches to be inserted in a special language. Complexity in the compiler is sometimes apparent with the addition of this type of language. Decreased checkout time for both programmer and machine may result by an appropriately designed patching language.

B. Source Language Level Patching

Can patches be made at the source language level? That is, is there a provision within the compiler which will allow small patches of source language level coding to be added? It is often convenient to try a certain patch which the programmer may think will work. If it doesn't or if he doesn't get the results that he wants he may then wish to pull that patch out and try a different one. Does the compiler system allow the programmer, once the patch is checked out, to automatically incorporate this in the program resulting in easy documentation? This feature can measurably decrease checkout time for the programmer.

C. Parameters in Design Point

The "Design Point" refers to the objectives, laid down by the designers, of what trade-offs have been made among compilation time, code efficiency, degree of diagnostic capability, size and capability of computing equipment used, etc. It is a specification of the intentions of the builders as to where emphasis is tailored to fit the local requirements. To what extent is the design point parameterized? State

the parameters involved, their effect, and the method of changing them. Identify each parameter and its limits relative to the possible objectives during construction. Product economy and growth flexibility are affected by this feature.

D. Disabling of Time Efficiency Trade-Offs

Is it easy to enable or disable the sections of the compiler reflecting time efficiency trade-offs? Describe how this can be done. Identify the flexibilities available for enabling and for disabling each and any compiler section which contributes to time efficiency trade-offs.

E. Efficiency of Object Code

A measure of efficiency could entail a comparison of code produced by the compiler with that produced by experienced machine language programmers. Efficiency would be measured in program operating time, amount of storage used and sophistication of the resulting program. Describe the method by which the above efficiency figure was estimated or calculated to involve all three possibilities. Does the compiler make a special pass as the last pass to optimize the object code produced?

F. Separation of Common Subfunctions

Are subfunctions common to more than one library routine explicitly separated into subroutines? It may be that certain subroutines within subroutines are common amongst several different library routines and thereby may be separated out in order to affect product economy. Determine the degree to which this is done.

G. Procedure for Running Large Programs (segmentation)

What procedures are available for running programs too large to fit in primary storage? Describe the segmentation scheme used and assess its flexibility. Is there an upper bound on the size of the program relative to the segmentation scheme? If large programs are under consideration a segmentation scheme is indeed necessary and will measurably decrease programming time and effort.

H. Common Data References for All Programs

Can all parts of the segmented programs above reference the same data? That is, does each segment in a large program have access to all of the data associated with the program? How is this implemented?

I. Modification of Program Control Sequence Dynamically

Can the sequence of control flow between parts of segmented programs be modified during program execution? A problem with segmented programs is that any given segment requiring control transfers to some other segment, finds it inefficient to loop back and forth. These segments may not always be in primary storage at the same time. However, it is necessary that any segment have the capability of getting to every other segment of the same program.

J. Automatic Library Search

Is there a procedure for automatically searching the library, and including all subroutines from it that are needed by the object program? If not, how are such subroutines to be included? If sufficient flexibility is provided for automatic library search then measurable decreases in programming time and effort may be realized.

K. Standard Record and File Description

Does the subroutine library contain standard record and file description? That is, is the library file of routines organized by using a standard method of identifying records or files within that library? This will allow other installations to use this library and therefore contribute to the machine independence of the compiler.

L. Compiler Table Sharing for Overflow

Are internal compiler tables arranged so that overflow from one may be placed in the unused space reserved for another? For example, say the compiler requires three different types of accumulative tables. For compilation of one type of program one of the tables needs a large amount of space whereas the other two need next to none. It is considered a judicious technique if one large area is reserved for use of these three combined tables such that any one can expand to the point where storage needed by all three combined will not exceed the overall storage allocation.

M. Dynamic Allocation of Storage

Is storage allocation in the compiler performed dynamically? That is, any tables or storage needed during the compilation process may be assigned dynamically in conjunction with the operating system such that primary storage is only used when needed. This feature will tend to reduce compilation time since more efficient use of the storage is made and thereby lessening requirements for segmenting or use of secondary storage.

N. Implementation of Storage Allocation Changes

How is a change in compiler storage allocation implemented? How difficult is it to make modifications to the allocation scheme or technique used within the compiler? Flexibility in this feature will tend to decrease time and effort in maintaining the compiler.

O. Hardware Configuration Changes at Programming Running Time

Can hardware configurations be changed easily for programs at object time? Is it relatively simple to implement changes at this time? Identify the limits of these changes in hardware configurations relative to each hardware component. This would tend to increase available execution time and decrease programming time in checkout; since if a program does not use certain hardware components, which are inactive for one reason or another at running time, a suitable configuration change can be effected in the programming system and the program allowed to run.

P. Combining Program Segments and Routines from other Software Packages

It is often desirable to include programs written for other purposes and possibly for other installations within this application. What facilities are available for combining program segments and routines produced for or by other software packages with those produced for or by this compiler? Is any of the required interfacing automatically performed? Assistance in combining and using these programs tends to increase machine independence and decrease programming and checkout time.

Q. Operation on Similar Machine Types

Can the output of a compilation run on more than one type of machine (other configurations of the same machine, other machines in the same series, machines of a different type through the use of an emulator, etc.)? Identify the machines and the configurations on which the output of this compiler may be executed. Flexibility provided by this feature will tend to enhance machine independence.

R. Utilization of Bit or Byte Accessing Facilities and Other Sophistications

Does the object code produced by the compiler make use of any bit or byte accessing facilities of the machine? Identify and describe any advantages the object code takes from whatever sophistication the hardware instructions set may provide. Efficient use of

hardware instruction sophistication tends to produce an object code which is economical in both execution time and storage required.

S. Packing Techniques for Conserving Storage

Describe the packing techniques used in the object code produced by the compiler to economize the storage utilization. Many compilers work strictly with full words and do not allow manipulation of bits or characters other than the use of special subroutines. How flexible is this compiler relative to packing bytes and bits into words? Both primary and secondary storage space is saved by appropriate use of this feature.

T. Truncation or Rounding of Conversions

Are the results of input conversions (especially floating point numbers) truncated or rounded from the true value? Can this be controlled by the user? Does it require compiler modifications? To what degree of precision are input conversions performed (e.g. BCD to binary number conversions)?

U. Subroutine Names Considered as Global Symbols

Global symbols are those defined throughout the entire program and do not change meaning from subroutine to subroutine. Subroutine linkage is the process of defining, during loading, the subset of global symbols comprised of subroutine names. There are two basic methods of handling such subroutine linkage: through a transfer vector (i.e. indirect) and by direct addressing. Are global symbol definitions implemented directly or indirectly? In the transfer vector method all subroutine calls are compiled as a call to a word before the first instruction of the calling program. There is one of these words for each subroutine called and the loader must modify only them. The advantage of this method is program loading flexibility and increased loading speed; however, transfer vectors occasionally account for more than 10% of all memory used. In the direct reference scheme, there are no transfer vector words and the loader must modify one word in the program for each instance of a program call. This method decreases memory requirements and execution time but may increase loading time. The indirect method will provide flexibility to the programmer and, therefore, save him time. The direct method will conserve execution time but combining programs or making modifications to programs will probably require machine language programming. The transfer vector scheme is a commonly used technique and therefore contributes to machine independence. The degree of each of these factors will depend on the proposed compiler.

V. Common Subexpressions Compiled Only Once

Are common subexpressions recognized so that code is compiled to compute them only once? Can the compiler recognize multiple occurrences of the same expression and compile code which will compute the expression only once and save the results that can be used in the place of computing the expression again? Effort by the compiler builder in this particular area will tend to produce economies in operating time and storage used in the object program.

W. Mathematically Equivalent Subexpressions

Does recognition of common subexpressions extend to mathematically equivalent subexpressions such as A*(B+C) and A*B+A*C? What types of expressions will be recognized as equivalent?

X. Modification of Constants

Can a program be written entirely in compiler language which will modify the value of a constant without causing a compilation, loading, or execution diagnostic error report? The programmer working only in compiler language should be protected against constants changing values. A way in which a constant value may be changed in some FORTRAN systems is the following:

```
SUBROUTINE COMPUTE (A)
  "
  "
A = 10
  "
  "
  "
END
```

If subroutine compute is now called with a constant argument, as in CALL COMPUTE (5.5), the value of the constant will be changed. If A was originally set by an input value to be constant throughout the program, erroneous results may occur because of the change. This feature will tend to reduce programmer and machine checkout time.

Y. Intermediate Level Languages Used

What is the target language of the compiler? What are the intermediate languages (if any)? Within some compilers the conversion is made from input language to an intermediate language as one step, and then intermediate language to object code in another step. This intermediate language provides a method for programmers to attach

other programs written separately in the intermediate language to a program written in the compiler language. This flexibility is often highly desirable since otherwise both portions of the problem would have to be done in the intermediate language. This feature will decrease programming and checkout time, and contribute to machine independence.

Z. Programmer Selection of Outputs

Can the programmer select and/or specify the outputs of a compilation? What options are available to him as to output including the original source language program, the intermediate language listing, the listing of the final object coding of the program, etc.? This feature will tend to decrease time spent in checkout for the programmer.

AA. Target Language Optimization Performed

Describe whatever target language optimization is performed. In particular, describe optimization in the following areas: elimination of duplicate constants, elimination of unnecessary store and/or access commands, unnecessary saving of index registers, etc. Judicious use of optimization techniques in this area will tend to produce an object code which is efficient both in overall execution time and in use of primary storage but may result in increased compilation time.

AB. Ease of Excluding Optimization

Since compilation time is often measurably increased by the use of optimizing techniques, it is often desirable to inhibit any optimizing done on the object code. Once the program is checked out, one final compilation run may be performed using the optimizing routines to insure production of an object program with efficient code. In case a compiler has routines that will perform optimization, this feature would be used to assess the flexibility provided for controlling optimization routines. Flexibility provided for programmers by this feature will tend to conserve compilation time.

IV. PROGRAMMER ERROR DIAGNOSTICS

A. Errors Causing Compilation Termination

Identify those source language errors which cause complete termination of compilation. Some programming errors are peculiar to the particular problem or application involved and as such are not detectable by the compiler. Others although detectable by the

compiler may be of minor importance and as such do not require termination of the compilation process. It must be determined exactly what errors will cause job termination or compilation termination. The most desirable situation would be one in which the programming system informs the operator or programmer (or both) that a major error has been detected but continues executing the normal job stream. Operator intervention, compilation time and execution time may each be affected in varying degrees.

B. Undetected Source Language Errors

Those source language errors which will not be detected either by the compiler or by any error checking program associated must be identified. Any conceivable error made in a source language program could be a contender in this list. Any undetected errors must be found eventually by the programmer thereby increasing his checkout time as well as compilation time. The concern here is to assess the completeness of the detected errors while fully realizing that many programming errors are detectable only by the programmer. Adequacy in this area will tend to decrease programmer checkout time.

C. Errors Causing Unexpected Stops

Identify any source language errors which may cause the computing system to halt or to delay with resulting lost facility time. Concentration here is on those errors which are both undetected and will cause unexpected stops. The cause may be software or hardware and may be in the central processor or a peripheral input device. The most desirable situation is one in which errors are detected and relayed to the operator and/or programmer with halts or stops completely controllable by them. Operator intervention is decreased and machine time utilization increased if very few or no such stops occur.

D. Other Source Program Errors

Identify those source program errors which are detected and describe them relative to the compilation phase in which they are detected. One extreme is the detection of all possible errors by the compiler in one pass thereby extending compilation time for errors which may rarely occur. That is, the compiler takes time to check for each situation and compilation time is increased. The other extreme is to have a bare minimum of errors detected, thereby extending programmer checkout time. The desire here is to meet a reasonable balance relative to the specific application. This balance must be determined by the user-evaluator team, and this feature would assist them in determining what effects would exist on programmer checkout time and compilation time.

E. Number of Errors Found Before Termination

Does the compiler attempt to find as many errors as possible or does it terminate after only one is found? The number and types of errors must be identified relative to each pass made by the compiler on the source language program. Programmer checkout time is saved if the programmer is provided as many appropriate diagnostics as possible each time he attempts compilation. On the other hand, compilation time is wasted if checks are made for a large number of errors when the first one detected was a major error and the others are directly affected by it.

F. Number of Passes Required for Errors

Does the compiler make one pass in which it finds all the related errors or is there several passes made in which each pass detects more complex and less obvious errors? What controls or parameters are available for selecting the number of passes? An excessive number of passes would increase compilation time unnecessarily. However, insufficient flexibility in the compilation process and in the diagnostic capability may result if only a single pass is performed.

G. Compiler Production After Error Detection

Will the compiler continue producing an object program after an error is found? If so, under what conditions? Identify each relevant error. This feature will affect compilation time in cases where compilation termination is performed only after a portion of the object program has been produced. Any processing which produces an object program is essentially wasted in this case. Is there some way the programmer can control the production of an object program?

H. Statement Identification for Errors

Does the diagnostic error report indicate the exact statement causing the error? How specific is the error report within the statement as to the nature of the error and its location in the statement? Time may be saved during checkout if diagnostic error reports are identified relative to the position within the program in which the error was found.

I. Detection of Excessive Source Language Symbols

What diagnostics are provided to indicate that the source program symbol table limits of the compiler have been executed. Checkout time may be saved for the programmer if this is detected during the compilation process. Also, significant amounts of compilation time may be saved by early detection (in the compilation process) of this condition.

J. Errors Detected During Execution

List the diagnostics provided when errors are detected during object program execution. Include a description of each as well as the procedure for restarting. Are specific hardware components identified for each appropriate diagnostic? Appropriate reporting of these errors to the operator and/or to the programmer may effect savings in execution time in addition to decreasing programmer checkout time.

K. Modifications for Trace Requests

What program modifications are required in order to insert requests for traces of program components, or periodic and sectional dumps? Identify those diagnostics which assist the programmer in inserting these types of requests accurately. A flexible scheme allowing quick but effective modifications to the program will result in decreased checkout time for the programmer and decreased execution time by the facility.

L. Prescan for Frequent Error Detection

Does the proposed compiler employ a special prescan to detect frequent errors? Is any error detection performed during initial read-in of the source program deck? Considerable compilation time may be saved by this special prescan providing the errors detected are those which have a reasonably high probability of occurring. This prescan process may be executed by the on-line or the off-line computer. At any rate, programmer and machine checkout time will be saved by a suitably designed prescan process.

V. HARD-COPY OUTPUT AND FACILITY DOCUMENTATION

A. Diagnostics Provided Off-Line

Are diagnostics indicating errors made by the object program recorded off-line for programmer information? Is sufficient information provided concerning each error so that the programmer may reconstruct the program path? Measures of software capability affected by this feature are programmer checkout time and documentation.

B. Automatic Update of Recompilations

Is there a procedure for modification of a previously compiled program without recompiling? Does this procedure provide for an automatic update of the source program? This feature will decrease compilation time as well as contribute to the documentation of the specific program.

C. Trace and Dump Outputs Printed

Are dump and trace outputs printed in source language? Is sufficient information provided in these printed outputs to allow the programmer easy reference to the source language program? This feature will tend to decrease programmer checkout time and also contribute to savings in time for effecting programming changes.

D. Trace Symbols and Statement Referencing

If dump and trace outputs are not printed in source language, how are the source program symbols and statements referenced in the dump and control flow trace outputs? What must the programmer do in order to connect his program in source language with these dump and flow trace outputs? Adequate provision of useful references will decrease programmer checkout time.

E. Accounting for Compiler Modifications

Describe the method used to keep track of modifications to the compiler. How can one identify the available features relative to any one version? It is desirable to have features relative to any one version identified explicitly which requires the use of a well-defined scheme. Both growth flexibility and documentation are enhanced by an appropriate and adequate scheme.

F. Documentation Intended for External Maintenance

Is documentation of the compiler intended to permit outside maintenance and modification? Is the documentation organized and cross referenced such that maintenance and modification is relatively easy? This will contribute to the measure of documentation.

G. Availability of Compiler Manuals and Listings

Which of the following documents are currently available:

(1) Listings of the Compiler.
(2) Manual of Compiler Operation.
(3) Manual of Compiler Internal Structure.
(4) Programmer's Reference Manual.
(5) Programmer Training Manual.

H. Outputs of Compilation and Loading

Which of the following listings are included in the output of compilation and loading:

(1) Listing of the Source Program.
(2) Listing of the Object Program.
(3) Cross References to Symbols and Statements.
(4) A List of All Data Areas and Their Locations.
(5) List of all Buffer Areas and Their Locations.
(6) A Storage Map Including Routines Loaded.
(7) Lists of All Interprogram. References.
(8) A List of All Errors Detected

VI. MANUAL AND AUTOMATIC OPERATIONS

A. Operator Message for Unavailable I/O Files

Is the computer operator informed whenever the required input/output files are not available? Is the message or messages given to the operator directed to him quickly so that he may make the appropriate files available? Efficient operator messages will decrease time spent on operator intervention.

B. Operator Messages for Object Program Errors

Are object program errors printed on-line along with directions to the operator for recovery or restart? Is the message relayed quickly and efficiently? Does the particular diagnostic make clear whether this requires complete job abortion or whether some simple procedure may be performed by the operator in order to continue. Efficiency in this area tends to reduce time spent on operator intervention.

C. Machine Error System Termination

Is the run terminated when a machine error occurs? Identify those conditions for which termination occurs. What special indicators are used to inform the operator of the termination?

D. Operator Options for Termination

Does run termination in the event of machine error occur at the discretion of the computer operator? If yes, is there sufficient information given to the computer operator to allow him to make this decision?

E. Information for I/O Device Malfunction

What information is furnished to the operator in case of an input/output device malfunction (e.g. magnetic tape)? Identify each case with the associated information furnished. Sufficient and appropriate information given to the operator will enhance I/O utilization as well as make operator intervention more efficient.

F. Failure Reports for Maintenance Personnel

Describe the procedures for providing maintenance personnel with equipment failure reports. Are reports for these messages continually being generated so that output will include past history reports? The point is to determine whether there is such a procedure and, naturally, how good it is. This feature will decrease time spent on maintenance.

G. Operator's Console Control of Trace Routines

Can the computer operator affect and control tracing routines and/or dump routines from his console? Identify the commands he may give. Describe the interaction with the programmer or with a specific program. Flexibility for operator control will tend to reduce program checkout time and operator intervention.

H. Programs for Updating Software System

Are programs available for updating the software system? That is, identify programs used for producing a new version of the system incorporating latest modifications. Describe how they interact with the system. Can they be operated in conjunction with other normal jobs? What limits exist for operation in the job queue? System maintenance is affected by this feature.

I. Minimum Configuration for Updating

What is the minimum hardware configuration required for this updating process? How does the minimum configuration relate to the proposed configuration? Maintenance is the main measure of software capability affected by this feature.

J. Operator Communication Via Console Keys

Can the program communicate with the console operator via the console keys and the on-line input/output equipment? In some installations, it is expedient to provide communication between the program and the console operator. To what degree can this be done? Although some savings in execution time may be experienced with adequate communication, the main saving is in decreased operator intervention requirements.

K. Time Required for Operator Actions

How much time is required for operator actions for an average compilation (e.g. tape loading)? Identify the actions required

and estimate overall times or times for each. Can these actions be done while another job is being run? To what limits? This feature tends to increase usable compilation and execution time as well as reduce operator intervention time.

L. Executive or Monitor Control Provision

Does the compiler operate within a separate monitor or executive system? If one is to be used, see Volume III of this report on Operating, Monitor or Executive Systems. If not, describe the operator functions incorporated within the compiler. Can the compiler be run without a separate operating system? Operator intervention, compilation time and execution are each affected by this feature. The presence of a well-designed set of operator functions tends to conserve all three measures of software capability.

M. Availability of Compile, Load and Execute Options

If no separate monitor or executive system is provided, what provisions are available within this compiler for selecting one or any combination of the following processes: compile, load, execute? Is there a special test-execute available? This feature will tend to decrease operator intervention time as well as increase the available execution and compilation time.

N. Flexibility of Compile, Load and Execute Options

Can the compiler or loader be called upon by an object program dynamically? What degree of flexibility does the programmer have in selection of any of the following processes: compilation, load, execute, test-execute, test data generation. Programming time tends to decrease with flexibilities in this area.

VII. FACILITY ADMINISTRATION

A. Same Compiler Version Used for Both Timing and Documentation

Is the compiler version, used for benchmark or similar types of computing system timing, the same version and modification described under all other features? This feature tends to decrease programmer checkout time and effort in maintaining a system.

B. Maintenance Performed by Vendor or User

Will the vendor maintain this compiler or will it be an additional function to the user? Will the user be getting a special

version of a compiler which is maintained by the vendor? If so, will the user be required to maintain all those portions of the compiler which differ from the basic one maintained by the vendor?

C. Stability of Compiler Design

Is the compilers design relatively stable? That is, is this compiler still in the stage where system programmers are developing better techniques to perform certain functions within the compiler or has this process been performed in the past resulting in a presently stable compiler? This feature will tend to reduce effort expended in maintenance.

D. Unusual Techniques of Implementation

Does the compiler employ any unusual techniques? Was it implemented in an unusual manner? Identify each unusual technique used or unusual implementation manner as to the effect it has on compilation time, execution time, complexity of the compiler, size of the compiler, compiler maintenance, etc.

E. Size of Machine Language Version

How large is the machine language version of the compiler? Size should be assessed in terms of percentage of available primary working storage used during an average compilation. If the compiler uses a very large percentage of available primary storage, then the compiler must compile smaller pieces of any given program and therefore take considerable more time to produce the required object code.

F. Use of Standard Modules Within Compiler

How much of the compiler consists of standard off the shelf modules? Is the compiler constructed in a modular fashion such that present and future requirements may be specified by appropriate selection of these modules? This feature will tend to make system program maintenance easier, and allow for growth flexibility.

G. Minimum Configuration Required for Compiler

What is the minimum hardware configuration required to run this compiler? If the configuration is different than the one proposed, describe the effects on the user-programmers, on program compilation time relative to this difference.

H. Performance Improvements from Larger Configurations

Describe how the use of a larger hardware configuration improves compiler and object program performance. If the user, in the

future, finds a need for a larger configuration, it would be desirable if the compiler could make use of the increased hardware.

I. Average Speed of Compiler (statements/minute)

What is the average speed of the compiler in statements per minute? This is a direct method of determining comparable compilation speeds.

J. Object Program Storage Space

How much object program storage space does the compiler allocate for fixed overhead, buffer areas, monitor system tables, etc.? Large amounts of space allocated by the compiler for these items will tend to reduce the efficiency of the object program and may affect secondary storage.

K. Command Structure Utilization

How much of the target computer's command structure is utilized by the code output from the compiler? List any modes or commands not used. Also, give reasons in the case of unused modes or commands. The concern here is to assess the degree to which the compiler uses the advantages and sophistication in the hardware instruction code.

SECTION FOUR

QUESTIONNAIRE

I. INTRODUCTION

The following questionnaire is intended as an aid to the evaluation of compilers. Those questions preceded by the word "COBOL" are to be answered only if COBOL is the principal software package proposed; similarly, questions preceded by the word "FORTRAN" are to be answered only if FORTRAN is the principal software package proposed. All other questions are always to be answered. It should be understood that all questions are assumed to refer to the principal software package and its compiler.

While many of the following questions are phrased in a manner implying a "yes" or "no" answer, this is not intended as a restriction on the respondents. Further explanation is encouraged whenever it leads to a clearer picture of the item in question; however, respondents are urged to remember that excessive detail can also hinder understanding. It is expected that most questions can be answered in a few hundred words or less.

Respondents should indicate appropriate references along with each answer and submit a copy of the referenced documents. In the event that no documentation for a given item exists, the name, title, and location of the individual or group responsible for the given area should be submitted.

II. FUNCTIONAL DIVISIONS

The questions are presented in seven sections:

A. SOURCE LANGUAGE ORIENTED FEATURES - questions relating primarily to characteristics of the language rather than the compiler.

1. (COBOL) What features of COBOL, Edition 1965, have not been included in the proposed compiler?

2. (COBOL) Describe any features this compiler has which provide increased capability or flexibility not included in the COBOL, Edition 1965 report.

3. (FORTRAN) List and describe all features of your language which are equivalent to some feature of the proposed ASA FORTRAN standard.

4. (FORTRAN) List and describe all features of your language which have no equivalent in the proposed ASA FORTRAN standard.

5. (FORTRAN) List and describe all features included in the proposed ASA FORTRAN standard but not included in your language.

6. Describe any non-standard implementation techniques. What effect do these have on the compilation speed and on the standards of the language?

7. Are source language facilities adequate to access all types of input/output equipment available for the machine?

8. Can dump and control flow trace requests be inserted at the source language level?

9. What languages are available on the proposed configuration?

10. What flexibility is available for varying the sizes of input/output records and of table sizes?

11. (FORTRAN) How do the restrictions on statement sequencing differ from those described in the proposed ASA FORTRAN standard?

12. (FORTRAN) Is the FREQUENCY statement accepted by the proposed compiler? If so, describe the effects of the FREQUENCY statement.

13. (FORTRAN) How do the rules for determining the definition of DO indices in the proposed compiler differ from those in the proposed ASA FORTRAN standard? How do the restrictions on subscript expressions and on expressions used as the limits of a DO differ from the proposed ASA FORTRAN standard?

B. LOGIC AND CONTROL OF INPUT/OUTPUT DEVICES - language and compiler features referring specifically to the control of input/output devices and associated program logic (i.e. buffer control, device assignment, etc.).

1. What are the restrictions on overlapping computation with input/output operations? Which of these restrictions are due to software rather than hardware?

2. What combinations of input/output units may be concurrently activated?

3. If the proposed hardware permits simultaneous input/output operations, can several input/output files be read or written simultaneously using the proposed software?

4. Can they be buffered simultaneously?

5. If the proposed configuration includes a priority interrupt system, does the proposed software system utilize this priority interrupt system for input/output processing?

6. Describe any options available in the area of input/output device servicing.

7. What are the advantages (in terms of ease of programming, economies in the object program, etc.) of the above options?

8. On which input/output devices will attempts to read or write be repeated if errors are detected?

9. How many attempts will be made on each device?

10. What is the basis for choosing the given number in each case?

11. How are the input/output subroutines supplied to the object program organized (by device, function, etc.)?

12. Which input/output subroutines are always loaded with the object program? Why?

13. If a particular device is needed (in the program under consideration) only for input, is the output routine also loaded? Why?

14. (COBOL) How many input/output areas have been implemented under the RESERVE option of the FILE-CONTROL statement?

15. (COBOL) When available buffer areas can hold more than one record, what special action is taken by the system?

16. If the proposed configuration includes a disc or drum, can the system simulate several sequential input/output devices (i.e. magnetic tapes) on a relatively random access device such as a disc?

17. (COBOL) Does the system use a random access device for sequential files defined in the DATA DIVISION? If so, can any single record be selected by a read or write verb?

18. (COBOL) If the proposed configuration includes a random access device (other than primary magnetic core storage), can this random access device be used for test data storage during debugging runs?

C. COMPILER FEATURES - features of the compiler (i.e. the implementation of the language) rather than the language itself.

1. Can patches be made at the source language level?

2. Is there a special patching language?

3. State the design point of the compiler and the major features which illustrate it.

   (Note: The "design point" refers to the objectives, laid down by the designers, of what trade-offs have been made among compilation time, code efficiency, degree of diagnostic capability, size and capability of computing equipment used, etc. It is a specification of the intentions of the builders as to where emphasis is tailored to fit the local requirements.)

4. To what extent is the design point parameterized? State the parameters involved, their effect, and the method of changing them.

5. Is it easy to enable or disable the sections of the compiler reflecting time efficiency trade-offs? How can this be done?

6. How efficient is the object code produced by the compiler?

   (Note: A measure of efficiency could entail a comparison of code produced by the compiler with that produced by experienced machine language programmers.)

7. Describe the method by which the above efficiency figure was estimated.

8. Are subfunctions common to more than one library routine explicitly separated into subroutines?

9. What procedures are available for running programs too large to fit in primary storage?

10. Can all parts of such programs reference the same data?

11. Can the sequence of control flow between parts of such programs be modified during program execution?

12. Is there a procedure for automatically searching a library and including all subroutines from it that are needed by the object program? If not, how are such subroutines to be included?

13. Does the subroutine library contain standard record and file descriptions?

14. Are internal compiler tables arranged so that overflow from one may be placed in the unused space reserved for another?

15. Is storage allocation in the compiler performed dynamically?

16. How is a change in compiler storage allocation implemented?

17. Can hardware configurations be changed easily for a program at object time? To what extent?

18. What facilities are available for combining program segments and routines produced by other software packages with those produced by the compiler? Is the required interfacing automatically performed?

19. Can the output of a compilation run on more than one type of machine (other configurations of the same machine, other machines in the same series, machines of a different type through the use of an emulator, etc.)? What machines?

20. Does the object code produced by the compiler make use of any bit or byte accessing facilities of the machine? Which ones?

21. Describe any packing techniques used in the object code produced by the compiler to economize the storage utilization.

22. To what degree of precision are input conversions (e.g. BCD to binary number conversions) performed?

23. Are the results of input conversions (especially to floating point representation) truncated or rounded from the true value?

24. Are global symbol definitions implemented directly or indirectly (transfer vector)?

25. Are subroutine names considered to be ordinary global symbols?

26. Are common subexpressions recognized so that code is compiled to compute them only once?

27. If so, over what range?

28. Does recognition of the above type extend to mathematically equivalent subexpressions such as A*(B+C) and A*B+A*C ?

29. Can a program be written entirely in compiler language which will modify the value of a constant without causing a compilation, loading, or execution diagnostic error report?

30. What is the target language of the compiler? What are the intermediate languages (if any)?

31. Can the programmer select and/or specify the outputs of a compilation?

32. Describe target language optimization performed. In particular, describe optimization in the following areas: elimination of duplicate constants, elimination of unnecessary store and access commands, elimination of the execution of unnecessary sub-formula evaluation in Boolean statements.

D. PROGRAMMER ERROR DIAGNOSTICS - questions intended to clarify a description of the facilities for error detection and reporting.

1. Which source language errors cause complete termination of compilation?

2. Which source language errors will not be detected by the compiler or by any error checking program?

3. Which source language errors can cause an unexpected halt at any point?

4. What other source program errors are detected and when are they detected?

5. Does the compiler attempt to find as many errors as possible or does it terminate after only one is found?

6. If it attempts to find many errors, does this happen in a single pass or may more be required?

7. Will the compiler continue producing an object program after an error is found? If so, under what conditions?

   (Note: For a user whose primary workload emphasizes the development of relatively small programs it is desirable that object code production be continued in order to facilitate detection of additional errors; however, if extremely large programs are being developed, object program production should be halted at the initial instance of an error to avoid wasting large amounts of machine time.)

8. Does the diagnostic error report indicate the exact statement causing the error? How specific is the error report within the statement as to the nature of the error and its location in the statement?

9. What diagnostics are provided to indicate that the limits of the compiler have been exceeded (e.g. too many source program symbols)?

10. List the diagnostics provided when errors are detected during object program execution. Include a description of each as well as the procedure for restarting.

11. Can dump and control flow trace requests be inserted without program modification?

12. Does the proposed compiler employ a special prescan to detect frequent errors?

13. Are frequent errors detected during the first pass of the compiler?

14. Is any error detection performed during initial read-in of the source program deck (possibly on a peripheral computer)?

E. HARD-COPY OUTPUT AND FACILITY DOCUMENTATION - questions designed to clarify the nature and frequency of documents produced by the compiler system (listings, error reports, etc.) as well as questions to determine what documentation about the compiler and the language is available.

1. Are diagnostics indicating errors made by the object program recorded off-line for programmer information?

2. If there is a procedure for modification of a previously compiled program without recompiling, does this procedure provide for an automatic update of the source program?

3. Are dump and trace outputs printed in source language?

4. If not, how are the source program symbols and statements referenced in the dump and control flow trace outputs?

5. Describe the method used to keep track of modifications to the compiler.

6. Is documentation of the compiler intended to permit outside maintenance and modification?

7. Which of the following documents are currently available?

   a) Listings of the Compiler.

   b) Manual of Compiler Operation.

   c) Manual of Compiler Internal Structure.

   d) Programmer's Reference Manual.

   e) Programmer Training Manual.

8. Which of the following are included in the output of compilation and loading?

   a) A Listing of the Source Program.

   b) A Listing of the Object Program.

c) Cross-References to Symbols and Statements in the Object Program.

d) A List of All Data Areas and Their Locations.

e) A List of All Buffer Areas and Their Locations.

f) A Storage Map Indicating Routines Loaded and Their Locations.

g) Lists of All Inter-Program References.

h) (FORTRAN) Lists of Variables Appearing in COMMON, DIMENSION and EQUIVALENCE Statements.

i) A List of All Errors Detected.

F. MANUAL AND AUTOMATIC OPERATIONS - primarily compiler and object system characteristics pertaining to running the compiler, updating to produce new compiler versions, etc.

1. Are object program errors printed on-line along with directions to the operator for recovery or restart?

2. Is run termination in the event of machine error at the discretion of the computer operator?

3. Is the run terminated when a machine error occurs?

4. Is the computer operator informed whenever the required input/output files are not available?

5. What information is furnished in the case of input/output device (e.g. tape) malfunction?

6. Describe the procedures for providing maintenance personnel with equipment failure reports.

7. Can dump and control flow trace routines be controlled from the operator's console?

8. Are programs available for updating the software system (producing a new version of the system incorporating latest vendor modifications)?

9. What is the minimum configuration required for this updating process and how long does it take?

10. Can the program communicate with the console operator via the console keys and the on-line input/output equipment?

11. How much time is required for operator actions for an average compilation (e.g. tape loading)?

12. Does the compiler operate within a monitor or executive system?

13. If so, describe the major functions of the interface.

14. What provisions are available for selecting one or any combination of the following processes: compile, load, execute? Can the sequence be selected in advance?

G. FACILITY ADMINISTRATION - questions pertaining to the administration of a facility using the given software package as well as those questions not conveniently included under some other category.

1. Is the compiler version used for timing the same version and modification described under all other features?

2. Who constructed the compiler? When?

3. Who maintains it?

4. Is compiler's design relatively stable?

   (Note: Frequent modifications in the design of a compiler tend to produce the situation of its being poorly checked out, whereas failure to modify compiler design over lengthy periods of time frequently results in performance substantially inferior to that possible with the current state-of-the-art.)

5. Does the compiler employ any unusual techniques? Was it implemented in an unusual manner?

6. How large is the machine-language version of the compiler?

7. How much of the compiler consists of standard off-the-shelf modules?

8. What is the minimum configuration required to run the compiler?

9. Describe how the use of a larger configuration improves compiler and object program performance.

10. In its present state, how many man-months of effort does the compiler represent?

11. What further effort is anticipated toward improving compiler performance?

12. What is the average speed of the compiler (statements per minute)?

13. What is the average line-to-line expansion ratio between target language and source language?

14. How much object program storage space does the compiler allocate for fixed overhead, buffer areas, monitor system code, etc.?

15. How much of the target computer's command structure is utilized by the code output from the compiler? List any modes/commands not used.

SECTION FIVE

COMPILER BENCHMARK PROGRAMS

I. INTRODUCTION

Data for comparative analysis has often been provided for evaluation groups doing equipment selection by using an actual programming problem typical of the application being considered. This problem may be specified in one of several ways. One way is to write a program in a language which is as independent as possible of any particular computing machine. This language, however, should be one for which compilers already exist. Another way is to describe the problem in a specification and ask the vendor to program it himself (in a language of his choosing) but run it with data generated by the evaluation team. These methods work adequately when the system functions to be performed by the equipment are well defined or previously specified and the procurement is sufficiently large to warrant the vendor's investment. These types of programs are sometimes called "application oriented" benchmarks.

Another type of benchmark is one which is oriented toward an entire category of data processing problems and can be termed "standard" since it depends on no single or particular system specification. Any number of problems exist which could be used as a test program.

II. GENERAL REQUIREMENTS

The following criteria summarize the essential benchmark selection requirements and may be used selectively within those limits or boundaries set by the particular problem under consideration.

A. Functional Performance

A program must perform some useful function containing realistic source language statements.

B. Unbiased

Each vendor should receive identical material.

C. Running Times

Compilation and execution times should be such that the program may be run in a small-to-medium speed computing system; however, expansion requiring only minimal reprogramming should enable the use of these programs for large system selections also.

D. Program and Data Size

The size of the program plus the associated data should be such that it does not exclude computing systems of small-to-medium storage capacities.

E. Reprogramming

Major reprogramming of the benchmark should not be required by the vendor.

F. Monitor Time

Overhead time of monitor or executive system functions should be a small percentage of the total.

G. Documentation

Program documentation must be complete including explanations of designer's intentions, listings of both the program and the data, flow charts, etc.

III. TEST PROGRAM FEATURES

A benchmark must test some subset of the features listed below. Previous sections have identified those features which may be determined best by the analytical approach. It should be noted that any feature tested with a benchmark may also be determined by using an appropriate questionnaire. The following features are those for which benchmark testing appears the most reasonable method:

A. Compilation Time Without Debugging Aids

Three cases must be considered when measuring compilation time. The first (compile only) occurs when execution does not immediately follow compilation (as in batch compiling to eliminate reloading of the compiler for each job); here compile time is the total time required to translate the source program (located on the system input unit) to an object program stored either on the system output unit or on some intermediate backup output unit. In the second case (compile-and-go) compilation time is the total time required to translate the source program into an object program stored in the computer and ready to be executed with the test data mounted. The third case (compile-and-load) is similar to the second except that no test data is needed. This option is used when the programmer requires a memory dump of the initial object program core allocation as well as the program compilation output. The system input unit and system output unit mentioned should be those normally used for job input and output. In multiprocessing and multiprogramming systems which overlap parts of compilation with other jobs, the compilation time statistics must include: (1) total real-time for job, (2) for each facility or component in the system the percentage of the total time during which it was unavailable to other jobs.

B. Execution Time Without Debugging Aids

Two execution times can be distinguished corresponding to the first two compile times above which may or may not include loading (the third case, compile and load, produces the same output for future execution purposes as does the compile only). The execution time is the time elapsed from the beginning (or end) of loading to the termination of the job, including the time required by any associated data channels to complete the transmission of data to and from the selected I/O units. Since use of debugging aids may substantially affect execution timing, this factor should be measured with and without them. This totals four different timings. Multiprogramming and multiprocessing considerations mentioned in the discussion of compilation time apply equally to execution timing.

C. Primary Storage Utilization by Compiler Without Debugging Aids

A memory map produced as part of the normal compilation output will provide appropriate information to calculate the percentage of primary storage used by the compiler. Generally, it is best to have the compiler occupy the smallest amount of storage possible.

D. Utilization of Peripheral Equipment by Compiler Without Debugging Aids

It is important to determine what specific equipments are used by the compiler. These may be observed visually during the benchmark demonstration. Relative ranking will depend on the specific user requirement.

E. Compilation Time With Debugging Aids

F. Primary Storage Utilization by Compiler With Debugging Aids

This point will differ from C. above in that the debugging programs may use portions of core. Again, the smaller percentage of storage used by the compiler, including the debugging programs, the better.

G. Utilization of Peripheral Equipment by Compiler With Debugging Aids

The differences between this feature and D. above lie with the implementation of the debugging programs. Peripheral equipment may be used differently or more extensively when compiling with these debugging programs. It is generally most desirable to utilize all available peripheral equipment. This feature will be visually observed.

H. Execution Time With Debugging Aids

I. Subroutine Linkage

There are two basic methods of handling subroutine linkage: through a transfer vector (indirect) and by direct addressing. In the transfer vector method all subroutine calls are compiled as a call to a word before the first instruction of the calling program. There is one of these words for each subroutine called and the loader must modify only those words. The advantage of this method is in increased loading speed; however, transfer vectors take up large amounts of storage if many subroutines are used. In the direct reference scheme there are no transfer vector words and the loader must modify one word in the program for each instance of a subroutine call. This method decreases memory requirements and execution time but may substantially increase loading time.

COBOL - What instruction set is generated whenever a PERFORM verb is used? Does the amount prohibit the utility for single statements

or short subroutines? Can the PERFORM verb be used with the INCLUDE verb (for library subroutines) without excessive code or undue compilations? Does the compiler recognize the existence of several PERFORM verbs on the same subroutine, thereby producing only one subroutine copy to avoid repetition? Is this linkage set up during compilation or during program loading?

FORTRAN - Global symbols are those defined throughout the entire program as opposed to local symbols, which are defined only in the subroutine within which they appear. Subroutine linkage is the process of defining, during loading, the subset of global symbols comprised of subroutine names.

J. Edited Output Listing

This listing will contain the memory map, locations of defined variables, listing of input-output subroutines used and library routines used. In some cases the listing provides a means of visually matching source language statements and resulting machine language coding. Generally, these features tend to ease the programmer's task by reducing program checkout time.

K. Cross-Referenced Symbol Listing

This listing might include each symbolic data element and the sequence number of every procedure statement which uses it. Also, branch points might be similarly tagged. Checkout at the source language level is made easier by this listing providing the debugging aids associated emphasize source language debugging. This type of listing is more often used by a COBOL compiler than by an algebraic compiler. At any rate this listing is valuable for reducing checkout time and is a desirable feature for this reason.

L. Existence of COMMON and EQUIVALENCE Statements (FORTRAN only)

M. Data for Trial Execution (COBOL only)

Are facilities available for generating and/or operating on trial files or other data to provide for compile-and-go runs. This allows a savings in checkout time over making up two separate runs.

N. Effects of FREQUENCY Statement (FORTRAN only)

To test this, two separate runs must be made - (1) without the FREQUENCY statement, and (2) with it. The comparison can then be made to see whether the compiler recognizes the statement and makes a corresponding change within the object program.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* The MITRE Corporation Bedford, Massachusetts	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

A METHOD FOR THE EVALUATION OF SOFTWARE, VOLUME II
Procedural Language Compilers - Particularly COBOL and FORTRAN

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
N/A

5. AUTHOR(S) *(Last name, first name, initial)*

Budd, Arthur E.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
April 1967	65	0

8a. CONTRACT OR GRANT NO. AF 19(628)-5165 b. PROJECT NO. 8510 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-66-113 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* MTR-197

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	EDP Equipment Office, Electronic Systems Division, L. G. Hanscom Field, Bedford, Massachusetts

13. ABSTRACT

This report contains procedural language compiler features considered important for comparative analysis. These features are identified in a form expressly for inclusion in the Three Step Method for Software Evaluation (Volume 1 of this series) under Category ONE: Procedural Language Compilers -- Particularly COBOL and FORTRAN.

DD FORM 1 JAN 64 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
ELECTRONIC DATA PROCESSING Evaluation Selection Software COBOL FORTRAN Procedural Language Compilers						

INSTRUCTIONS

I. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

GPO 886-551